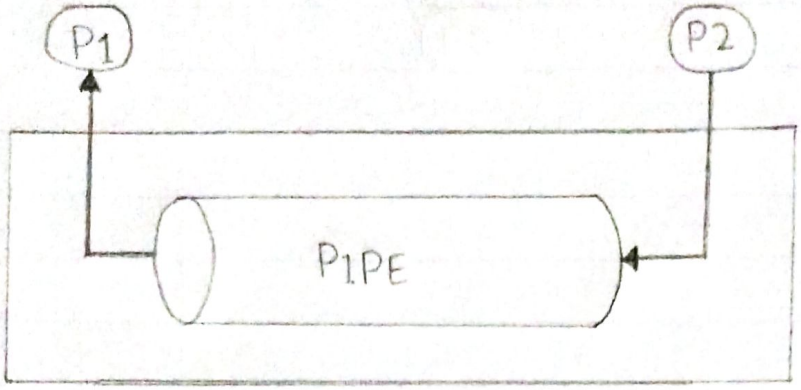
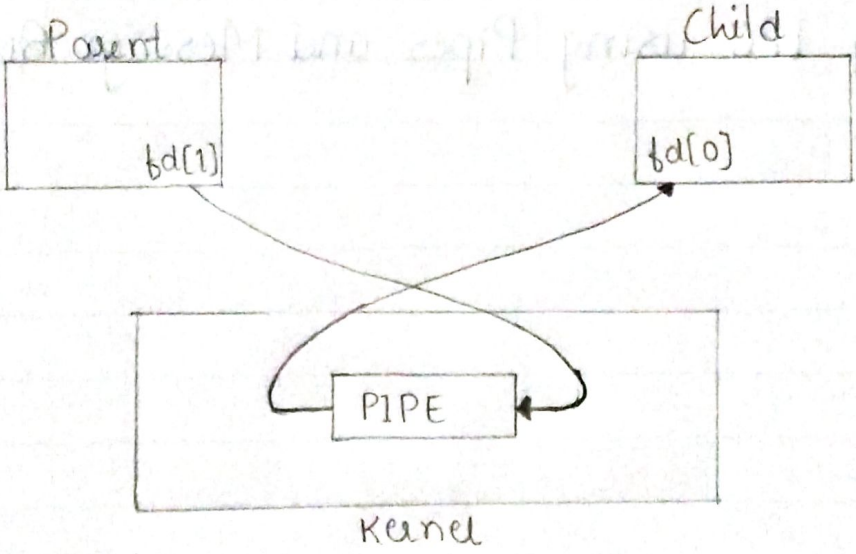
TERMWORK - 1

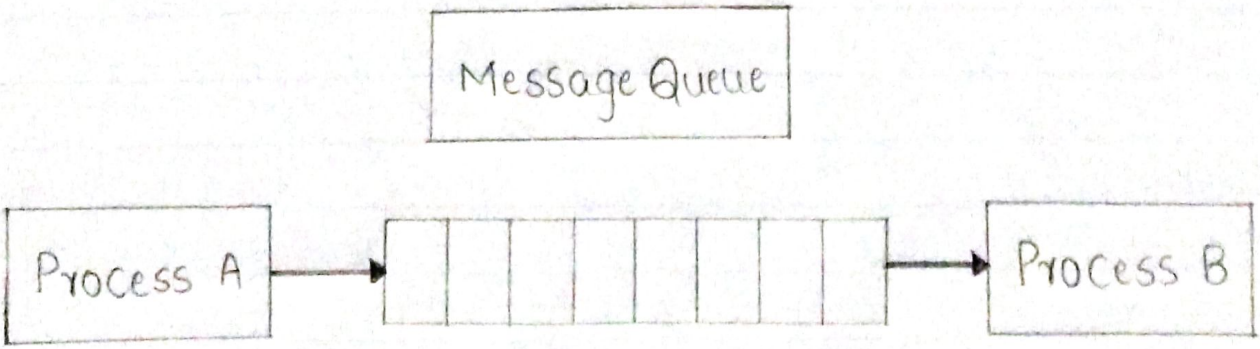
Implementing IPC using Pipes and Message Queue

IPC - Pipes and Message Queue



Communication between parent and child





OBJECTIVES

1. To understand inter-process communication and how to implement it using different form.
2. To Know how to implement ipc using pipes and message queue
3. To Know how pipe and message queue works

THEORY :

IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multi-processing systems. There are various forms of IPC, they are pipes, message queues, semaphores etc.

Pipe is a mechanism by which the output of one process is directed into the input of another process. Thus it provides one way flow of data between two related processes. One can write into pipe from input end and read from the output end. A pipe descriptor, has an array that stores two pointers, one for its input end and other for its output end.

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by msgget(). New messages are added to the end of the queue by msgsnd(). Messages are fetched from a queue by msgrcv().
All processes can exchange information through access to a common system message queue.

ALGORITHM :

Using Pipes :
Step 1 : Create a pipe
Step 2 : Create a child process
Step 3 : Parent process writes to the pipe
Step 4 : Child process retrieves the message from the pipe and writes it to the standard output

Using Message Queue
Step 1 : Create a message queue or connect to an already existing message queue (msgget())
Step 2 : Write into message queue (using writer process)
Step 3 : Read from the message queue (using reader process)
Step 4 : Perform control operations on the message queue.

(Here we will create 2 processes, one to write and another to read)

SOURCE CODE :

PIPE :

```
#include<sys/wait.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>

#define ReadEnd 0
```

```
#define WriteEnd 1

void report_and_exit (const char * msg ){
      perror (msg);
      exit(-1);
}

int main() {
     int pipeFDs [2] ;
     char buf ;
     const char * msg = "Nature's first green is gold\n" ;
     if (pipe (pipeFDs) < 0) report_and_exit ("pipeFD") ;
     pid_t cpid = fork() ;
     if ( cpid < 0) report_and_exit ("fork") ;

     if ( 0 == cpid ) {
          close (pipeFDs [WriteEnd]) ;
          while (read (pipeFDs [ReadEnd], &buf, 1) > 0)
              write ( STDOUT_FILENO, &buf, sizeof (buf)) ;
          close (pipeFDs [ReadEnd] ;
          _exit (0) ;
     }
     else {
          close (pipeFDs [ReadEnd]) ;
          write (pipeFDs [WriteEnd], msg, strlen (msg)) ;
          close (pipeFDs [WriteEnd]) ;
          wait (NULL) ; exit (0) ;
     }
     return 0 ;
}
```

Message Queue :
Writer Process

```
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<stdlib.h>
#define MAX 10

struct mesg_butter{                    //structure for message queue
      long mesg_type;
      char mesg_text[100];
}message;

int main(){
    key_t key;
    int msgid;
    key = ftok("progfile", 65);
    msgid = msgget(key, 0666|IPC_CREAT);
     message.mesg_type = 1;
     printf("Write Data : ");
     fgets(message.mesg_text, MAX, stdin);
     msgsnd(msgid, &message, sizeof(message), 0);
     printf("Data send is : %s \n", message.mesg_text);
      return 0;
}
```

Reader Process

```
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/msg.h>

struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main() {
    key_t key;
    int msgid;   key = ftok("progfile", 65);
    msgid = msgget(key, 0666|IPC_CREAT);
    msgrcv = (msgid, &message, sizeof(message), 1, 0);
    printf("Data Received is : %s \n", message.mesg_text);
    msgctl(msgid, IPC_RMID, NULL)
    return 0;
}
```

CONCLUSION :

In this termwork, we've discussed about inter-process communication, it's various forms i.e. pipes and message queue and implementation of pipe and message queue.

REFERENCES :

1. www.geeksforgeeks.org/ipc-technique-pipes/
2. www.geeksforgeeks.org/ipc-using-message-queues/
3. www.tutorialspoint.com

OUTCOMES :

At the end of experiment, we were able to

1. understand inter-process communication and its implementation
2. understand implementation of pipe and message queue

2GI18CS142

TERMWORK-2

Implementing client server communication using socket programming that uses connection oriented protocol at transport layer

OBJECTIVES :

1. To know about connection protocol oriented protocol i.e. TCP at transport layer
2. To know about socket, its type and working.
3. To implement client server communication using socket programming.

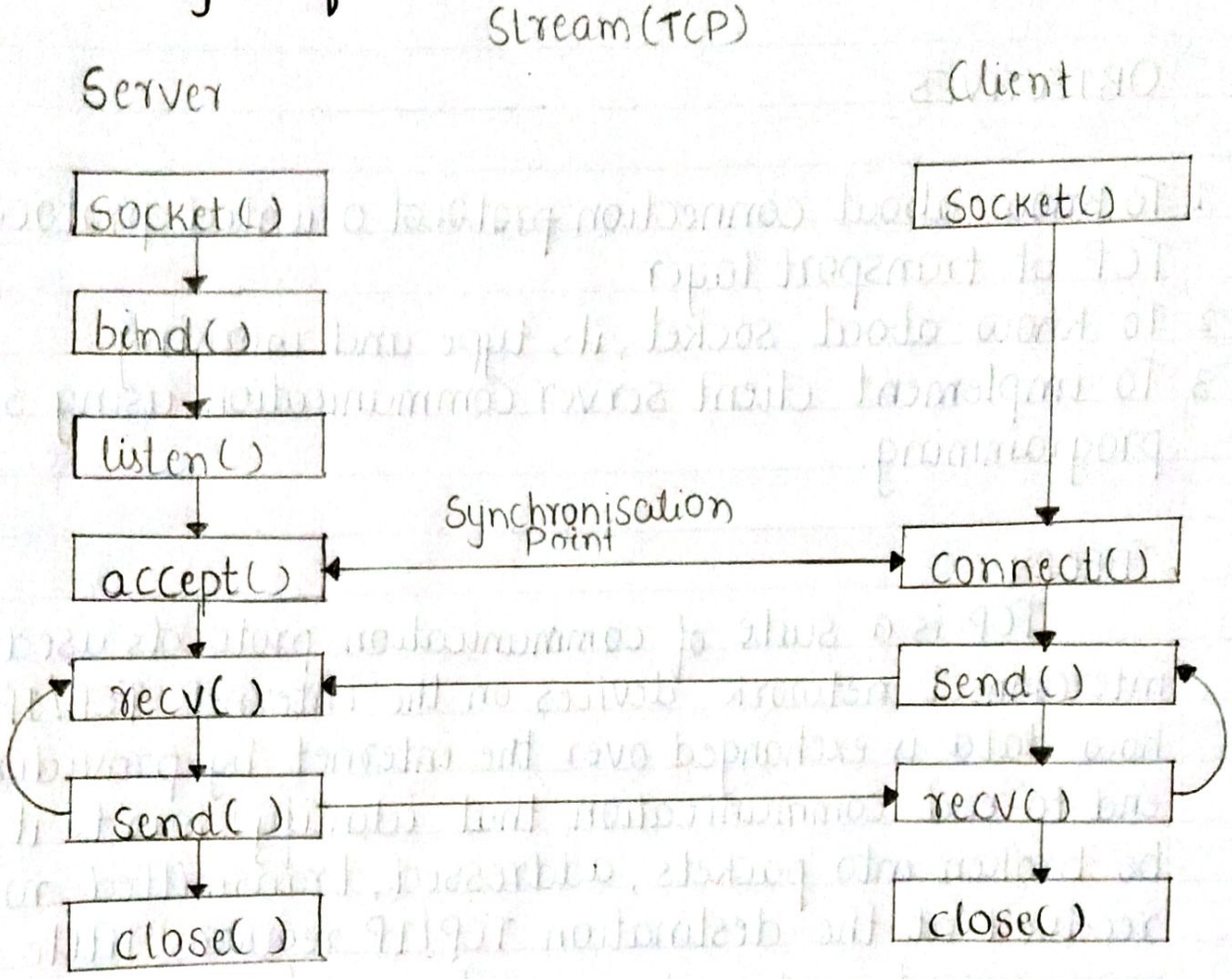
THEORY :

TCP is a suite of communication protocols used to interconnect network devices on the internet. TCP/IP specifies how data is exchanged over the internet by providing end-to-end communication that identify how to it should be broken into packets, addressed, transmitted, routed and received at the destination. TCP/IP requires little central management and is designed to make networks reliable with the ability to recover automatically from the failure of any device on the network.

A socket is one endpoint of a two way communication link between two programs running on network. The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication takes place.

Client-Server Communication involves 2 components, namely a client and a server. They are usually multiple clients in communication with a single server. Client initiates the communication, sends requests to the server. It is active socket. Client must know the address and the port of the server. Server passively wait for and responds to clients. It is passive socket

State Diagram for Server and Client Model.



ALGORITHM :

Server

Step 1 : using Create(), Create TCP socket

Step 2 : using bind(). Bind the socket to server address

Step 3 : using listen(), put the server socket in a passive mode, where it waits for the client to approach the server to make a connection

Step 4 : using accept(), At this point, connection is established between client and server, and they are ready to transfer data

Step 5 : Go back to step 3

Client

Step 1 : Create TCP socket

Step 2 : Connect newly created client socket to server

SOURCE CODE :

Server.c

```
#include<stdio.h>
#include<netinet/in.h>
#include<netdb.h>
#include<sys/types.h>
#include<unistd.h>
#define SERV_TCP_PORT 5035

int main(int argc, char ** argv){
        int sockfd, newsockfd, clength;
        struct sockaddr_in serv_addr, cli_addr;
```

```
char buffer[4096];
sockfd = socket(AF_INET, SOCK_STREAM, 0);
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(SERV_TCP_PORT);
printf("\n Start");
bind(sockfd,(struct sockaddr *)&serv_addr,
        sizeof(serv_addr));
printf("\n Listening....");
printf("\n");
listen(sockfd,5);
clength = sizeof(cli_addr);
newsockfd = accept(sockfd,(struct sockaddr *)&cli_addr,
                    &clength);
printf("\n Accepted");
printf("\n");
read(newsockfd, buffer, 4096);
printf("\n Client message : %s", buffer);
write(newsockfd, buffer, 4096);
printf("\n");
close(sockfd);
return 0;
}
```

Client.c

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<netdb.h>
#define SERV_TCP_PORT 5035

int main(int argc, char *argv[]){
    int sockfd;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    char buffer[4096];
    char buf[4096];
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    serv_addr.sin_family = AF_INET;
    // inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    serv_addr.sin_port = htons(SERV_TCP_PORT);
    printf("\n Ready for sending.... ");
    connect(sockfd, (struct sockaddr*) &serv_addr,
                    sizeof(serv_addr));
    printf("\n Enter the message to send \n");
    printf("\n Client : ");
    fgets(buffer, 4096, stdin);
```

```
    write (sockfd, buffer, 4096);
    read (sockfd, &buf, 4096);
    printf ("Server echo Rxd : %s", buf);
    printf ("\n");
    close (sockfd);
    return 0;
}
```

CONCLUSION :

In this termwork, we've discussed about TCP/IP connection, sockets, client server communication and implementation of a client-server communication using socket programming.

REFERENCES :

1 www.geeksforgeeks.org/tcp-server-client-implement-in-c
2 www.geeksforgeeks.org/socket-in-computer-network
3 www.tutorialspoint.com/inter_process_communication

OUTCOMES :

At the end of the experiment, we were able to

1 understand connection oriented protocol i.e. TCP, socket and its working
2 implement client-~~serving~~ server communication using socket programming